Production and Rewrite Systems

Horatiu Cirstea, Claude Kirchner, Michael Mm Moossen, Pierre-Etienne Moreau

MANIFICO Méta-compilation non intrusive du filtrage par contraintes
Non intrusive meta-compilation of matching by constraints

Sous Projet 1
RETE et réécriture

Production and Rewrite Systems

Description :	This report studies the relationship between producton systems and term rewrite systems.
Auteur(s) :	Horatiu CIRSTEA, Claude KIRCHNER, Michael MOOSSEN, Pierre-Etienne MOREAU
Référence :	MANIFICO / Sous Projet 1 / Fourniture 1.2 / V0.8
Date :	31 août 2004
Statut :	à valider
Version :	0.8

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de l'économie, des finances et de l'industrie

ILOG, INRIA Lorraine, INRIA Rocquencourt

Historique

31 août 2004	V 0.8	création du document
15 septembre 2004	V1.0	version finale

Contents

Introduction

There is a strong and somewhat renewed interest in production and business rule systems. For example it is significant that in April 2004 Gardner published a comparative study of the different business rule systems and vendors and estimated that the current market of 200 millions of dollars was promised to a sustained increased for several years.

Production systems, as we will call them from now on, are widely appreciated in industry for their agility, reactivity and flexibility. But putting them in action is also delicate since their semantics is unclear or even unknown and therefore make the development of large systems quite depend of test and experience.

Term rewriting is a different kind a rule based system that emerged from automated deduction and semantics of programming languages in the early seventies. It is now a pretty well understood concept and many implementations show their usefulness, robustness and efficiency.

This report intends to bridge these a priori different views of rule based systems.

Some landmarks on production systems are:
– the initial RETE (Latin for 'net') algorithm and its use in OPS5 [For82, For81],
– an alternative, the TREAT approach [Mir90, MBLG90],
– The work by François Fages and its collaborators at Thomson leading to initial version of ilog-RULE: [FL91, FL92]
– the first link between production rules and rewriting done by Snyder and Schmolze [SS96],
– production rules as strategic rewriting in ELAN: [Dub01, DK99a, DK99b, DK00b, DK00a]
– with concerns about the verification of rule based system, for example the verification of production rules applied to raw steel manufacturing process [KDK93],
– and emerging from the constraint (logic) programming community, Constraint Handling Rules, CHR, a language for defining constraint solvers, but at the same time it is one of the most powerful multiset rewriting languages [Frü98]. – and also real time approaches like in [FL91, LG89, Lop87].

This leaded to the emergence of many implementations, including: G2 used in chemical industry, BladeAdvisor, CLIPS, ILOG-Rule, RulesPower commercialized by Forgy, LibRT selling a business rule checker called `Valens`, X-Tra a French ops5 implementation written in lisp [X-T88], Jess a CLIPS implementation in Java, JeOPS a OPS5 implementation in Java.

1 Definition of production systems

This section recalls the main concepts and definitions as introduced in [CKMM04].

1.1 Informal presentation of production systems

A production system consists mainly of the following five components:

- The Fact Types are user defined datatypes, like structs with fields or properties. There are intended for organizing the data that will be manipulated, for instance, we can have a *fact type* representing a house with properties like color, price, availability, and so on. But, notice that in most cases, we are restricted to *basic types* for the properties, so it could not be possible to have a property of type address in the house fact type defined before, if the address is a composed data type.

  We can then view a *fact* as a concrete assignment of values to the properties for a given fact type, for instance, an *available red* house that costs *one thousand*.

- The Working Memory ($\mathcal{WM}$) is the current program state, it is a global structure of facts. We will see later that this structure could be implemented either by sets or multisets.

- Production Rules are conditional statements of the form

  [Name] if Condition then Action

A rule has a name and it acts by addition and retraction of facts on the $\mathcal{WM}$ iff the *condition* is fulfilled. Here the *condition* is usually called the left hand side (LHS) of the production rule and the *action* its right hand side (RHS). The *condition* may or not be satisfied by the $\mathcal{WM}$ as described in the next section together with more precise explanation for *condition* and *action*. When the LHS of a rule is satisfied, the rule is said to be *activated*.

- The Production Memory ($\mathcal{PM}$) is the set of production rules, also known as Knowledge Base. It is almost always unvarying, in spite of some production system implementations that provide facilities to manipulate the production memory as RHS actions.

- A Resolution Strategy that consists of an algorithm for selecting just one rule to execute, if the conditions of the LHS of more than one rule are satisfied at the same time.

The production system interpreter executes a production system by performing a sequence of operations called *recognize-act cycle*:

1. Match: evaluate the LHS of each rule to determine which ones are activated given the current state of the $\mathcal{WM}$. This is the most time consuming step in the execution of a production system, and here is were the rete algorithm is used.

2. Conflict Resolution: select one activated rule. If no rule is activated, halt the interpreter returning the current state of the $\mathcal{WM}$.

3. Act: perform the actions specified in the RHS of the selected rule.

4. go to step 1.

When a rule is activated, an instantiation[1] is generated as an ordered pair of the form:

<rule, list of facts that satisfy its LHS>.

Instantiations are maintained in the *Conflict Set* ($\mathcal{CS}$). Then, the *Resolution Strategy* selects just one rule of this set, and its RHS is executed; it is said that the rule is fired.

A schematic view of the data work flow in a production system[2] is shown in Figure 1.

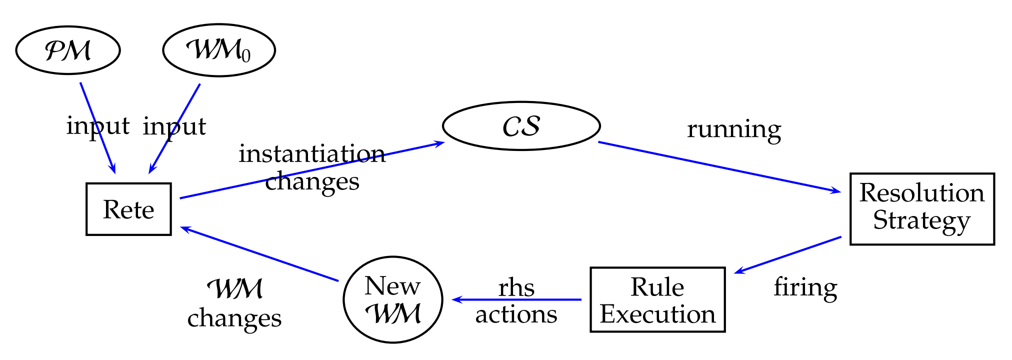


Figure 1: production system work flow

[1]this is a historical name, that does not reflect the common meaning of instantiation
[2]With non-modifiable Production Memory

1.2 General Production systems

We consider a set $\mathcal{F}$ of function symbols, usually denoted $f, g, h, \ldots$, a set $\mathcal{P}$ of predicate symbols, and infinite sets $\mathcal{X}$ and $\mathcal{L}$ respectively called set of variables and of labels. Variables are denoted $x, y, z, \ldots$ In most of the practical situations, finite set of labels are enough. These sets are assumed to be disjoint. Each function symbol and predicate symbol has a fixed arity. Nullary function symbols are called *constants*. We assume that there is at least one constant. The set of *terms* (denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$), *ground terms* (denoted $\mathcal{T}(\mathcal{F})$), *atomic propositions*, *litterals* (i.e. atomic proposition or their negation), *propositions*, *sentences* (i.e. closed propositions) are defined as usual in term rewriting [KK99, BN98, "T02] and first-order logic [Gal86].

We will freely use the usual notion of substitution. Notice that since in general first order propositions are instantiated, the substitution mechanism works modulo alpha-conversion to take care of the variable bindings.

Definition 1.1 A *fact* f is a ground term, $f \in \mathcal{T}(\mathcal{F})$.

Definition 1.2 We call *working memory* ($\mathcal{WM}$) a set of facts, i.e. it is a subset of the Herbrand universe defined on the signature.

Definition 1.3 A *production rule* or simply *rule* or *production*, denoted

$$[l]\ \texttt{if}\ p, c\ \texttt{remove}\ r\ \texttt{add}\ a$$

consists of the following components.

- A name from the label set: $l \in \mathcal{L}$.
- A set of positive or negative *patterns* $p = p^+ \cup p^-$ where a *pattern* is a term $p_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and a negated pattern is denoted $\neg p_i$. p^- is the set of all negated patterns and p^+ is the set of the remaining patterns.
- A proposition c whose set of free variables is a subset of the pattern variables: $Var(c) \subseteq Var(p^+)$.
- A set r of terms whose instances could be intuitively considered as intended to be removed from the working memory when the rule is fired, $r = \{ r_i \}_{i \in I_r}$, where $Var(r) \subseteq Var(p^+)$.
- A set a of terms whose instances could be intuitively considered as intended to be added to the working memory when the rule is fired, $a = \{ a_i \}_{i \in I_a}$, where $Var(a) \subseteq Var(p^+)$.

Such a rule is also denoted $[l]\ p, c \Rightarrow r, a$.

Remark: Indeed in the previous definition, one can discuss the choice of set as the data structure to represent, add and remove facts.

Definition 1.4 Given a set of facts $\mathcal{S}$ and a set of positive patterns p^+, p^+ is said to *match* $\mathcal{S}$ with respect to a theory $\mathcal{T}$ and a substitution σ, written $p^+ \ll^{\sigma}_{\mathcal{T}} \mathcal{S}$ if:

$$\forall p \in p^+ \quad \exists t \in \mathcal{S} \mid \sigma(p) =_{\mathcal{T}} t$$

We say that a set of negative patterns p^- dis-matches a set of facts $\mathcal{S}$, written $p^- \not\ll_{\mathcal{T}} \mathcal{S}$ iff:

$$\forall \neg p \in p^- \quad \forall t \in \mathcal{S} \quad \forall \sigma \mid \sigma(p) \neq_{\mathcal{T}} t$$

Definition 1.5 Given a substitution σ, a production rule $[l]\ p, c \Rightarrow r, a$ is $(\sigma, \mathcal{WM}')$-*fireable* on a working memory $\mathcal{WM}$ when

1. $p^+ \ll^{\sigma}_{\mathcal{T}} \mathcal{WM}'$
2. $p^- \not\ll_{\mathcal{T}} \mathcal{WM}$
3. $\mathcal{T} \models \sigma(c)$

for a minimal (with respect to the subset ordering) subset $\mathcal{WM}'$ of $\mathcal{WM}$. A fireable rule is also called an *activation*.

Definition 1.6 Given a production rule $[l]\ p, c \Rightarrow r, a$ which is $(\sigma, \mathcal{WM}')$-fireable on a working memory $\mathcal{WM}$, its *application* leads to the new working memory $\mathcal{WM}''$ defined as:

$$\mathcal{WM}'' = (\mathcal{WM} - \sigma(r)) \cup \sigma(a)$$

This is denoted $\mathcal{WM} \Rightarrow^{\sigma,\mathcal{WM}'}_{[l]\ p,c\Rightarrow r,a} \mathcal{WM}''$ or simply $\mathcal{WM} \Rightarrow \mathcal{WM}''$. The couple $(\sigma(r), \sigma(a))$ is called the $(\sigma, \mathcal{WM}')$-*action* of the production rule $[l]\ p, c \Rightarrow r, a$ on the working memory $\mathcal{WM}$.

Definition 1.7 For a given working memory $\mathcal{WM}$ and a set of production rules $\mathcal{R}$, the set

$$\mathcal{CS} = \{\ (l,\ \sigma) \mid \exists\ ([l]\ p, c \Rightarrow a, r)\ \in\ \mathcal{R} \text{ which is } (\sigma, \mathcal{WM}')\text{-fireable on } \mathcal{WM}\ \}$$

is called the $\mathcal{R}@\mathcal{WM}$-*conflict set*

A conflict set could be either empty (no rule is fireable), unitary (only one rule can fire), finite (a finite number of rule is activated) or infinitary (an infinite number of matches could be found due to the theory modulo which we work [FH86]). Whether finite or infinite, one should decide which rule should be applied: this is one of the major topics of interest in production systems, addressed by *resolution strategies*.

Definition 1.8 A *resolution strategy* is a computable function that given a set of production rules $\mathcal{R}$, and a production derivation

$$\mathcal{WM}_0 \Rightarrow \mathcal{WM}_1 \Rightarrow \ldots \Rightarrow \mathcal{WM}_n$$

returns a unique element of the $\mathcal{R}@\mathcal{WM}_n$-conflict set.

We have now all the ingredients to provide a general definition of production systems:

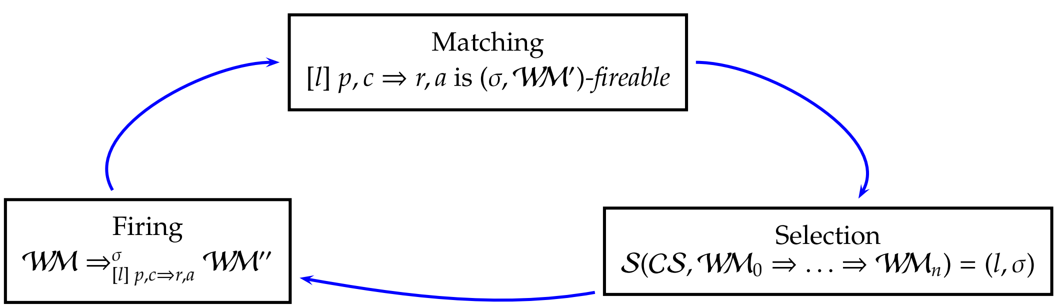
Definition 1.9 A *general production system* is defined as

$$\mathcal{GPS} = (\ \mathcal{F},\ \mathcal{P},\ \mathcal{X},\ \mathcal{L},\ \mathcal{WM}_0,\ \mathcal{R},\ \mathcal{S},\ \mathcal{T}\ )$$

Where:

- $\mathcal{F}$ is the set of function symbols,
- $\mathcal{P}$ is the set of predicate symbols,
- $\mathcal{X}$ is the set of variables,
- $\mathcal{L}$ is the set of labels,
- $\mathcal{WM}_0$ is the initial working memory,
- $\mathcal{R}$ is the set of production rules over $\mathcal{H} = (\ \mathcal{F},\ \mathcal{P},\ \mathcal{X}\ \mathcal{L}\ )$,
- $\mathcal{S}$ is the resolution strategy,
- $\mathcal{T}$ is the matching theory.

Definition 1.10 The *Inference Cycle* is defined as follows:



Remark: This definition of a production system is quite general as the pattern can be deep and non-linear, the condition can be an arbitrary first-order proposition, the resolution strategy can take the full derivation history into account.

In order to get good performances when running production systems, suitable restrictions have been used for the so called rete algorithm.

2 Production rules as objects

Production systems can be implemented from an object oriented point of view. This is detailed in the thesis of Hubert Dubois [Dub01].

3 Production rules as AC-rewrite systems

3.1 Snyder and Schmolze's approach

We summarize here the approach proposed by Snyder and Schmolze in [SS96].

The three basic components of a Production System:

- The working memory
- The collection of production rules
- The interpreter for applying rules repeatedly to the memory, subject to some conflict resolution strategy.

are represented, using the main notion of associative and commutative terms, in the following way.

Working Memory The memory is considered to be a set[3] of positive ground atoms, completed with negatives according to the *Reiter's Closed World Assumption*.

$$W = P \cup N$$

where:

- P is a finite set of ground atoms (the actual memory facts), and
- $N = \{\neg A | A \notin P\}$ (all other negative facts)

Production Rules Depending on the fact that a production rule contains a *Negation-As-Failure* (*NAF*) condition[4] or not, it may be translated into two different types of rewrite rules:

- Preserving rewrite rules.
  For production rules without a *NAF* condition, has the form:

  $$L_1, \ldots, L_n \longrightarrow L'_1, \ldots, L'_n$$

  where on each side of the arrow we have sets of literals (possibly containing variables) and where:

  $$\forall i \; L'_i = L_i \vee L'_i = \bar{L}_i$$

  In other words, the set of atoms is the same on each side, but the sign of a particular atom may have flipped from positive to negative (representing a deletion), from negative to positive (representing an addition), or has remained the same (representing a side condition). The application of such a rewrite rule preserves the property of being a working memory.

  All production rules that only test for membership, only add or delete facts, and do not have *NAF* conditions, can be formalized by one or more preserving rewrite rules. [5]

[3] The Snyder and Schmolze's full paper also considers the use of a multiset
[4] A negative condition which contains a variable which appears nowhere else.
[5] More than one rule may be necessary, since a rule might add an atom without testing for its absence

- Constrained preserving rules.
  This kind of rules have the form $S \longrightarrow T[\![\varphi]\!]$ where $S \longrightarrow T$ is a preserving rewrite rule and φ is a first-order formula where every free variable in φ occurs in S (and thus in T). However, we only need constraints of a particular form; roughly, we will transform each *NAF* condition L_i involving *NAF* variables $x_{i,1}, \ldots, x_{i,m_i}$ into a subconstraint of the form $\varphi_i = \forall x_{i,1}, \ldots, x_{i,m_i} L_i$, where none of the $x_{i,j}$ appear in $S \setminus \{L_i\}$. This results in a constraint of the form $[\![\varphi_1 \wedge \cdots \wedge \varphi_n]\!]$ for a production rule with n *NAF* conditions.

  But some *NAF* variable $x_{i,j}$ may be constrained by some condition C in L_i, if this is the case, the constraint φ_i should be expanded to $\varphi_i \vee \neg C$.

Production Rule Interpreter We formalize the last main feature of a production rule system as a rewriting process subject to checking the constraints.

We say that a WM W rewrites to W' with respect to a set of constrained rules R, denoted $M \longrightarrow_R M'$, if

- there exists a rule $P \longrightarrow T[\![\varphi]\!]$ in R,
- M has the form $M\{P\sigma\}$ for some substitution σ such that $M \models \varphi\sigma$, and
- M' has the form $M\{T\sigma\}$.

The subset $P\sigma$ is called the *redex*.

Given this, we have a good approximation of a production rule system as an AC-Rewrite system, but we do not model the conflict resolution strategy.

Since we are acting on term rewriting (in this case modulo AC), the resolution of conflicts could be done in at least two ways.

The first one consists of having the production system encoded as a confluent AC-rewrite system. In this case, there is always a common reduct for any two terms issued from a common ancestor. If moreover the rewrite system is terminating, the result is uniquely defined, independently of the rewrite strategy in use. AC-term rewriting and its meta-properties have been extensively studied, from AC-unification [Sti75], AC-completion [PS81, Hul80] (generalizing Knuth and Bendix approach [KB70]), to AC-termination [RN93, Rub02]. Moreover, AC-matching [Hul79, Eke95] can be efficiently compiled [KM01]. Even if the theoretical background of confluent and terminating rewrite system is now well investigated, this approach has the drawback that, indeed, only few production systems are in essence confluent and terminating.

The second one consists in using explicitly rewrite strategies for explicitly guiding the term rewriting process. This is typical of the approach taken in the ELAN system where rewrite rule are split in two categories. The first one called “un-labeled rules” are used for computation and are therefore assumed to form a confluent and terminating rewrite system. The second kind of rules are “labeled rules” and are assumed to be controlled by a strategy using the labels of the rule to fire the appropriate one. This is described in particular in [KKV94, BCD+04] and gave rise to the rewriting calculus [CK01], generalizing term rewriting as well as lambda-calculus and where strategy guided rewriting can be given a precise semantics [CKLW03]. Moreover (symbolic) constraint rewriting could be simply described in this setting, providing an explicit handling of substitutions and exceptions [FK02, CFK04] but also of structure with sharing and cycles [BBCK04].

For a complete example of such an encoding, see the appendix Section 6.7 page 18.

3.2 Using matching constraints

We model a production system using an approach simplifying the one by Schmolze and Snyder. The main originality is the introduction of a new predicate “don’t match” which is true iff a condition does not match any fact in the working memory. It is therefore a dis-matching problem that shares many similarities with dis-unification one [Com91].

Definition 3.1 Let s, t be terms and T a theory. A matching constraint, denoted $s \ll^?_T t$, has a solution σ when $\sigma(s) =_T t$. When a matching constraint is unsatisfiable (i.e. has no solution), this is denoted $s \not\ll_T t$: there exist no substitution σ such that $\sigma(s) =_T t$. When T is the empty theory, it is just omitted.

A production system can be then modeled is the following way:
Propositions are considered as terms of type bool.
The working memory is a term, denoted $\mathcal{WM}$ and built using the constructors:

- nil
- an associative and commutative operator "," with neutral element nil.

A production rule $[l]$ `if` p, c `remove` r `add` a is a conditional rewrite rule

$$c, r \to a \text{ if } C$$

with the standard semantics of AC-rewriting, that is:

$$\mathcal{WM} \to_{c \to a \text{ if } C} \mathcal{WM}' \Leftrightarrow \exists \sigma, \sigma(c) =_{AC} \mathcal{WM} \text{ and } \mathcal{WM}' = \sigma(a), \text{ provided } \sigma(C)$$

See Section 6.7.2 on page 18 for a simple example of this encoding.

4 Production rules via Constraint handling

After discussing with HAK [ck: Mar 8 04]
Conditions are constraints
Actions are constraints of the form $x' = f(x)$, where x' is the next value of x.
After computing the value $f(x)$, one gets the constraint: $x = oldValue \wedge x' = newValue$.

Then one needs a meta operation that

1. remove the constraint ($x = oldValue \wedge x' = newValue$) from store
2. add the constraint $x = newValue$ to store
3. propagate (may be not yet)

In this way x gets its new value
a bit (too?) magic i.e. too much adhoc??

Another try:

1. remove the constraint $x = oldValue$ from store
2. add the constraint $x = x'$ to store
3. propagate (to identify x and x' everywhere, but propagate will do more. . .)

5 Criticism of these previous approaches

All do not take care of the main pb of the rule firing semantics, i.e.

- when becomes a simple rule fire-able?
- could the user specify the order in which (s)he wants the fire-able rules to be fired?
- when unspecified what is the firing order?

Acknowledgments: Thanks to François Charpillet for sharing with us his X-tra experience.

6 Appendix: Examples

This appendix shows a same toy example, first in abstract notation and then implemented in several production system.

6.1 Abstract Notation

```
P := {   not,   <,    =,   ^   }

F := {
   house_id: house -> number,
   color: house -> atom,
   price: house -> number,
   forrent: house -> atom {true|false},
   houseaddress_id: houseadress -> number,
   houseaddress_number: houseaddress -> number,
   houseaddress_street: houseaddress -> string,
   houseaddress_city: houseaddress -> string,
   myaddress_number: myaddress -> number,
   myaddress_street: myaddress -> string,
   myaddress_city: myaddress -> string,
   side1: war -> string,
   side2: war -> string
}

W0 := {
   (house house_id()=1 ^ color()=red ^ price()=341 ^ forrent()=true),
   (houseaddress houseaddress_id()=1 ^ houseaddress_number()=251 ^
              houseaddress_street()="Rue Jeanne D'Arc" ^ houseaddress_city()=Nancy),
   (house house_id()=2 ^ color()=blue ^ price()=390 ^ forrent()=true),
   (houseaddress houseaddress_id()=2 ^ houseaddress_number()=121 ^
              houseaddress_street()="Avenue de Brabois" ^ houseaddress_city()=Villers-les-Nancy),
   (house house_id()=3 ^ color()=red ^ price()=415 ^ forrent()=true),
   (houseaddress houseaddress_id()=3 ^ houseaddress_number()=31 ^
              houseaddress_street()="Rue Carnot" ^ houseaddress_city()=Vandoeuvre-les-Nancy),
   (myaddress myaddress_number()=2551 ^ myaddress_street()=Gorbea ^ myaddress_city()=Santiago),
   (searching),
   (war side1()=usa side2()=irak)
}

R := {
  if
    (searching) ^
    (house_id(?h)=?id ^ color(?h)=red ^ price(?h)<400 ^ forrent(?h)=true) ^
    (houseaddress_id(?ha)=?id ^ houseaddress_number(?ha)=?num ^ houseaddress_street(?ha)=?str ^
       houseaddress_city(?ha)=?cit) ^
    (myaddress) ^
    not (war side1()=france) ^
    not (war side2()=france)
  then
    remove (searching)
    forrent(?h)=false
    myaddress_number(myaddress)=?num ^ myaddress_street(myaddress)=?str ^ myddress_city(myaddress)=?cit
  end if
}

P := { false/0, true/0, not/1, </2, =/2, ^/2, remove/1, add/1 }

F := { house/4, houseaddress/4, myaddress/3, war/2, searching/0 }

W0 := {
   house(1, red,341, true),   houseaddress(1, 251, "Rue Jeanne D'Arc", Nancy),
   house(2, blue,390, true),  houseaddress(2, 121, "Avenue de Brabois", Villers-les-Nancy),
   house(3, red,415, true),   houseaddress(3, 31, "Rue Carnot", Vandoeuvre-les-Nancy),
   myaddress(2551, Gorbea, Santiago),
   searching(),
   war(usa, irak)
}
```

```
R := {
  if
    searching() ^
    house(?id, red, ?pr, true) ^
    ?pr < 400 ^
    houseaddress(?id, ?num, ?str, ?cit) ^
    myaddress(?mn, ?ms, ?mc) ^
    not war(france, ?s2) ^
    not war(?s1, france)
  then
    remove(searching()),
    remove(house(?id, red, ?pr, true)),
    add(house(?id, red, ?pr, false)),
    remove(myaddress(?mn, ?ms, ?mc)),
    add(myaddress(?num, ?str, ?cit))
  end if
}
```

6.2 OPS5

OPS5 Core Syntax Explanations

- `;` : line comment.
- `(literalize <factname> <slotname>*)`: used for defining a new type of fact called `<factname>`, with 0, 1 or more slots (or properties) `<slotname>*`.
- `(p <rulename>? <pattern>* --> <action>*)`: used for defining a rule called `<rulename>`, with conditions or patterns `<pattern>*` and actions `<action>*`.
- `(make <factname> [^<slotname> <value>]*)`: used for creating a new fact, called `<factname>`, adding it to the WM.
- `(remove <cid>)`: used for retracting the fact that satisfy the condition given by `<cid>`, which is just the number of the condition in the LHS.
- `(modify <cid> (^<slotname> <newvalue>)+)`: used for modifying the fact that satisfy the condition given by `<cid>`. Modifying just the values of the given slots by `<slotname>` replacing the value with `<newvalue>`. Internally implemented as a `(remove <cid>)` followed by a `(make ...)`.

OPS5 Example

```
;new type(fact) definitions
;a fact called house(representing a house) with 4 "properties": id, color, price and forrent
(literalize house id color price forrent)
;a fact called houseaddress(representing the address of a house) with 4 "properties":
;id, number, street and city
(literalize houseaddress id number street city)
;a fact called myaddress(representing my address) with 3 "properties": number, street and city
(literalize myaddress number street city)
;a fact called searching with no properties, just for indicating that i am looking for a new house!
(literalize searching)
;a fact called war(representing a war between 2 sides) with 2 "properties": side1 and side2
(literalize war side1 side2)

;a rule with the name search_for_house
(p search_for_house
;if the searching flag is set
  (searching)
;search for a red house with a price less than 400
  (house ^id <id> ^color red ^price < 400 ^forrent true)
;get the address of that house
  (houseaddress ^id <id> ^number <num> ^street <str> ^city <cit>)
;get my address
  (myaddress)
```

```
;if there is no war where france is involved
 -(war ^side1 france)
 -(war ^side2 france)
-->
;i found a nice house so i delete the searching fact
  (remove 1)
;modify the state of the house
  (modify 2 ^forrent false)
;modify my address
  (modify 4 ^number <num> ^street <str> ^city <cit>)
)

;working memory initialization
;first house
(make house ^id 1 ^color red ^price 341 ^forrent true)
(make houseaddress ^id 1 ^number 251 ^street "Rue Jeanne D'Arc" ^city Nancy)
;second house
(make house ^id 2 ^color blue ^price 390 ^forrent true)
(make houseaddress ^id 2 ^number 121 ^street "Avenue de Brabois" ^city Villers-les-Nancy)
;third house
(make house ^id 3 ^color red ^price 415 ^forrent true)
(make houseaddress ^id 3 ^number 31 ^street "Rue Carnot" ^city Vandoeuvre-les-Nancy)
;my address
(make myaddress ^number 2551 ^street Gorbea ^city Santiago)
;searching flag
(make searching)
;war!
(make war ^side1 usa ^side2 irak)
```

6.3 CLIPS

CLIPS Core Syntax Explanations

- `;` : line comment.
- `(deftemplate <factname> (slot <slotname>)*)`: used for defining a new type of fact called `<factname>`, with 0, 1 or more slots (or properties) `<slotname>*`.
- `(defrule <rulename>? ([<cid> <- ]?<pattern>)* => <action>*)`: used for defining a rule called `<rulename>`, with conditions or patterns `<pattern>*` which can be associated to a given condition id `<cid>`, and actions `<action>*`.
- `(assert <factname> (<slotname> <value>)*)`: used for creating a new fact, called `<factname>`, adding it to the WM.
- `(deffacts <factgroupname> (<factname> (<slotname> <value>)*)*)`: used for creating several facts at once.
- `(retract <cid>)`: used for retracting the fact that satisfy the condition given by `<cid>`.
- `(modify <cid> (<slotname> <newvalue>)+)`: used for modifying the fact that satisfy the condition given by `<cid>`. Modifying just the values of the given slots by `<slotname>` replacing the value with `<newvalue>`. Internally implemented as a `(retract <cid>)` followed by a `(make ...)`.

CLIPS Example

```
;new type(fact) definitions
;a fact called house(representing a house) with 4 "properties": id, color, price and forrent
(deftemplate house (slot id) (slot color) (slot price) (slot forrent))
;a fact called houseaddress(representing the address of a house) with 4 "properties":
;id, number, street and city
(deftemplate houseaddress (slot id) (slot number) (slot street) (slot city))
;a fact called myaddress(representing my address) with 3 "properties": number, street and city
(deftemplate myaddress (slot number) (slot street) (slot city))
;a fact called searching with no properties, just for indicating that i am looking for a new house!
```

```
(deftemplate searching)
;a fact called war(representing a war between 2 sides) with 2 "properties": side1 and side2
(deftemplate war (slot side1) (slot side2))

;a rule with the name search_for_house
(defrule search_for_house
;if the searching flag is set, and bind the matching fact to ?f1
  ?f1 <- (searching)
;search for a red house with a price less than 400, and bind the matching fact to ?f2
  ?f2 <- (house (id ?id) (color red) (price ?p) (forrent true))
  (test (< ?p 400))
;get the address of that house
  (houseaddress (id ?id) (number ?num) (street ?str) (city ?cit))
;get my address, and bind the matching fact to ?f3
  ?f3 <- (myaddress)
;if there is no war where france is involved
  (not (war (side1 france)))
  (not (war (side2 france)))
=>
;i found a nice house so i delete the searching fact
  (retract ?f1)
;modify the state of the house
  (modify ?f2 (forrent false))
;modify my address
  (modify ?f3 (number ?num) (street ?str) (city ?cit))
)

;working memory initialization
(deffacts startup
;first house
  (house (id 1) (color red) (price 341) (forrent true))
  (houseaddress (id 1) (number 251) (street "Rue Jeanne D'Arc") (city Nancy))
;second house
  (house (id 2) (color blue) (price 390) (forrent true))
  (houseaddress (id 2) (number 121) (street "Avenue de Brabois") (city Villers-les-Nancy))
;third house
  (house (id 3) (color red) (price 415) (forrent true))
  (houseaddress (id 3) (number 31) (street "Rue Carnot") (city Vandoeuvre-les-Nancy))
;my address
  (myaddress (number 2551) (street Gorbea) (city Santiago))
;searching flag
  (searching)
;war!
  (war (side1 usa) (side2 irak))
)
```

6.4 JeOPS

JeOPS Core Syntax Explanations

- First you have to define one Java class for each type of Fact. They have not to implement or extend nothing special. And you can use any method for accessing they fields, ie: public attributes, getters and setters, or whatever.
- Then the rules definitions are described in a especial class which has to be compiled by jeops, with extension `.rules`. This class should define all the rules, and each rules should have three parts:
  - `declarations`: Here you have to specify the type of the involved facts, giving a variable name to each one. This step already does matching. If there are no facts in the WM of a required fact type, the rule is not executed.
  - `conditions`: In this section you can specify several Java conditions, using the variables defined in the previous section. There is also a section called `preconditions` which can be additionally used.
  - `actions`: in this section you can execute actions using the following methods: `retract(FactObject)`, `assert(FactObject)` and `modified(FactObject)`, the last one should be explicit called

after you modify any relevant information of the given `FactObject`, calling indirectly the rete algorithm.

- And at last, you have to define the main class which should instantiate an instance of the knowledge base generated by JeOPS, and initialize the WM by creating and asserting some Facts, for finally calling the `run()` method for execution.

JeOPS Example

```
// definition of Fact classes
public class House {
  public int id;        public String color;
  public double price;  public boolean forRent;
}
public class HouseAddress {
  public int id, number;
  public String street, city;
}
public class MyAddress {
  public int number;
  public String street, city;
  public String toString() {
    return ""+number+", "+street+", "+city;
  }
}
public class Searching {}
public class War {
  public String side1, side2;
}

//the rules knowledge base
public ruleBase HousesBase {
  rule SearchForHouse {
    declarations
      Searching searching;
      House house;
      HouseAddress houseAddress;
      MyAddress myAddress;
      War war;
    conditions
      house.forRent;
      house.color.equals("red");
      house.price<400.0;
      house.id == houseAddress.id;
      !war.side1.equals("france");
      !war.side2.equals("france");
    actions
      retract(searching);
      house.forRent=false;
      modified(house);
      myAddress.number=houseAddress.number;
      myAddress.street=houseAddress.street;
      myAddress.city=houseAddress.city;
      modified(myAddress);
  }
}

//main program, initializing the WM and running
public class TestHouses {
  public static void main(String[] args) {
    HousesBase kb = new HousesBase();

    House h1 = new House();
    h1.id=1;h1.color="red";h1.price=341.00;h1.forRent=true;
    HouseAddress ha1 = new HouseAddress();
    ha1.id=1;ha1.number=251;ha1.street="Jeanne D'Arc";ha1.city="Nancy";
    kb.assert(h1);
    kb.assert(ha1);
```

```
    House h2 = new House();
    h2.id=2;h2.color="blue";h2.price=390.00;h2.forRent=true;
    HouseAddress ha2 = new HouseAddress();
    ha2.id=2;ha2.number=121;ha2.street="Avenue de Brabois";ha2.city="Villers-les-Nancy";
    kb.assert(h2);
    kb.assert(ha2);

    House h3 = new House();
    h3.id=3;h3.color="red";h3.price=415.00;h3.forRent=true;
    HouseAddress ha3 = new HouseAddress();
    ha3.id=3;ha3.number=31;ha3.street="Rue Carnot";ha3.city="Vandoeuvre-les-Nancy";
    kb.assert(h3);
    kb.assert(ha3);

    MyAddress ma = new MyAddress();
    ma.number=2551;ma.street="Gorbea";ma.city="Santiago";
    kb.assert(ma);

    Searching s = new Searching();
    kb.assert(s);

    War w = new War();
    w.side1="USA";w.side2="Irak";
    kb.assert(w);

    System.out.println("i lived here:\n" + ma);
    kb.run();
    System.out.println("and moved to:\n" + ma);
  }
}
```

6.5 JRules, using TRL

Using the same fact classes defined in the JeOPS example.

Working Memory Initialization

```
assert  [  ]  [  ]  House  [  ]
    so that  id  =  1
    and  color  =  "red"
    and  price  =  341
    and  forRent  =  true
assert  [  ]  [  ]  HouseAddress  [  ]
    so that  id  =  1
    and  number  =  251
    and  street  =  "Rue Jeanne D Arc"
    and  city  =  "Nancy"
assert  [  ]  [  ]  House  [  ]
    so that  id  =  2
    and  color  =  "blue"
    and  price  =  390
    and  forRent  =  true
assert  [  ]  [  ]  HouseAddress  [  ]
    so that  id  =  2
    and  number  =  121
    and  street  =  "Avenue de Brabois"
    and  city  =  "Villers-les-Nancy"
assert  [  ]  [  ]  House  [  ]
    so that  id  =  3
    and  color  =  "red"
    and  price  =  415
    and  forRent  =  true
assert  [  ]  [  ]  HouseAddress  [  ]
    so that  id  =  3
    and  number  =  31
    and  street  =  "Rue Carnot"
    and  city  =  "Vandoeuvre-les-Nancy"
assert  [  ]  [  ]  MyAddress  [  ]
```

```
        so that  number  =  2551
        and  street  =  "Gorbea"
        and  city  =  "Santiago"
    assert  [   ]   [   ]  Searching  [   ]
        [so that]
    assert  [   ]   [   ]  War  [   ]
        so that  side1  =  "USA"
        and  side2  =  "Irak"
```

Knowledge Base

```
WHEN
    there is a   [   ]  Searching  [   ]   [   called  ?f1  ]
        [where]
        [such that]
    there is a   [   ]  House  [   ]   [   called  ?f2  ]
        where  id  is called  ?id
        such that  color    =  red
              and  price    <  400
              and  forRent  =  true
    there is a   [   ]  HouseAddress  [   ]   [   called  ?f3  ]
        [where]
        such that  id  =  ?id
    there is a   [   ]  MyAddress  [   ]   [   called  ?f4  ]
        [where]
        [such that]
    there is no   [   ]  War  [   ]
        [where]
        such that  side1  =  france
               or  side2  =  france
THEN
    retract  ?f1
    modify   [   ]  ?f2
        so that  forRent  =  false
    modify   [   ]  ?f4
        so that  number  =  ?f3.number
            and  street  =  ?f3.street
            and  city    =  ?f3.city
ELSE
```

6.6 ELAN

```
module houses

import global bool builtinInt;
end

sort Object Space
    House HouseAddress MyAddress Searching War;
end

operators global
 @ U @                        : (Space Space) Space (AC);
 empty                        : Space;
 @                            : (Object) Space;

 House[h_id=@, h_color=@, h_price=@, h_forrent=@]
                              : (builtinInt builtinInt builtinInt builtinInt) House;
 @                            : (House) Object;

 HouseAddress[ha_id=@, ha_number=@, ha_street=@, ha_city=@]
                              : (builtinInt builtinInt builtinInt builtinInt) HouseAddress;
 @                            : (HouseAddress) Object;

 MyAddress[ma_number=@, ma_street=@, ma_city=@]
                              : (builtinInt builtinInt builtinInt) MyAddress;
 @                            : (MyAddress) Object;

 Searching[]                  : Searching;
```

```
 @                            : (Searching) Object;

 War[w_side1=@, w_side2=@]    : (builtinInt builtinInt) War;
 @                            : (War) Object;

 go                         :  builtinInt;
 result(@)                   : (Space) builtinInt;
 occursWar1(@,@)              : (Space builtinInt) bool;
 occursWar2(@,@)              : (Space builtinInt) bool;
end

stratop global
  loop   : <Space> bs;
end

rules for Space
 S : Space;
 id1, id2, pr  : builtinInt;
 num, str, cit : builtinInt;
 num2, str2, cit2 : builtinInt;
global
 [housesearch]
    S U House[h_id=id1, h_color=1, h_price=pr, h_forrent=1]
      U HouseAddress[ha_id=id2, ha_number=num, ha_street=str, ha_city=cit]
      U MyAddress[ma_number=num2, ma_street=str2, ma_city=cit2]
      U Searching[]
 =>
    S U House[h_id=id1, h_color=1, h_price=pr, h_forrent=0]
      U HouseAddress[ha_id=id2, ha_number=num, ha_street=str, ha_city=cit]
      U MyAddress[ma_number=num, ma_street=str, ma_city=cit]
    if id1 == id2
    if pr < 400
    if not(occursWar1(S,1))
    if not(occursWar2(S,1))
 end
end

rules for builtinInt
 S : Space;
 num, str, cit : builtinInt;
global
 [] go => result(S)
    where S:=(loop) empty
             U House[h_id=1, h_color=1, h_price=341, h_forrent=1]
             U HouseAddress[ha_id=1, ha_number=251, ha_street=1, ha_city=1]
             U House[h_id=2, h_color=2, h_price=390, h_forrent=1]
             U HouseAddress[ha_id=2, ha_number=121, ha_street=2, ha_city=2]
             U House[h_id=3, h_color=1, h_price=415, h_forrent=1]
             U HouseAddress[ha_id=3, ha_number=31, ha_street=3, ha_city=3]
             U MyAddress[ma_number=2551, ma_street=4, ma_city=4]
             U Searching[]
             U War[w_side1=2,w_side2=3]
 end

 [] result(S U MyAddress[ma_number=num, ma_street=str, ma_city=cit]) => num       end
end

rules for bool
 S : Space;
 n, v : builtinInt;
global
 [] occursWar1(S U War[w_side1=n,w_side2=v],n) => true  end
 [] occursWar1(S,n)                            => false end
 [] occursWar2(S U War[w_side1=v,w_side2=n],n) => true  end
 [] occursWar2(S,n)                            => false end
end

strategies for Space
implicit
  [] loop => repeat*(first one(housesearch))   end
```

end

end

6.7 AC-Rewriting

6.7.1 Snyder and Schmolze's approach

$P := \{false/0, true/0, not/1, < /2, = /2,^{/} 2\}$
$F := \{house/4, houseaddress/4, myaddress/3, war/2, searching/0\}$
Wp := {
house(1, red,341, true), houseaddress(1, 251, ”Rue Jeanne D'Arc”, Nancy),
house(2, blue,390, true), houseaddress(2, 121, ”Avenue de Brabois”, Villers-les-Nancy),
house(3, red,415, true), houseaddress(3, 31, ”Rue Carnot”, Vandoeuvre-les-Nancy),
myaddress(2551, Gorbea, Santiago), searching(), war(usa, irak)
}
$W_0 := Wp \cup \{A | \neg A \in Wp\}$
R := {
searching(),
house(?id, red, ?pr, true), ¬house(?id, red, ?pr, false),
houseaddress(?id, ?num, ?str, ?cit),
myaddress(?mn, ?ms, ?mc), ¬myaddress(?num, ?str, ?cit),
¬war(france, ?s2), ¬war(?s1, france)
$\longrightarrow$
¬searching(),
¬house(?id, red, ?pr, true), house(?id, red, ?pr, false),
houseaddress(?id, ?num, ?str, ?cit),
¬myaddress(?mn, ?ms, ?mc), myaddress(?num, ?str, ?cit),
¬war(france, ?s2), ¬war(?s1, france)
$[\![\forall ?s1, ?s2 \quad \neg\text{war(france, ?s2)}, \neg\text{war(?s1, france)} \quad \wedge \quad \exists ?pr \quad ?pr < 400]\!]$
}

6.7.2 Using matching constraints

We follow here the methodology and encoding presented in section 3.2 on page 8.

$\mathcal{F} := \{false/0, true/0, not/1, < /2, = /2,^{/} 2\} \cup \{house/4, houseaddress/4, myaddress/3, war/2, C, searching/0\}$
W_0 := {
house(1, red,341, true), houseaddress(1, 251, ”Rue Jeanne D'Arc”, Nancy),
house(2, blue,390, true), houseaddress(2, 121, ”Avenue de Brabois”, Villers-les-Nancy),
house(3, red,415, true), houseaddress(3, 31, ”Rue Carnot”, Vandoeuvre-les-Nancy),
myaddress(2551, Gorbea, Santiago), searching(), war(usa, irak)
}
R := {
$?P\ U\ searching()$
$U\ house(?id, red, ?pr, true)$
$U\ houseaddress(?id, ?num, ?str, ?cit)$
$U\ myaddress(?mn, ?ms, ?mc)$
$\longrightarrow$
$?P$
$U\ house(?id, red, ?pr, false)$
$U\ houseaddress(?id, ?num, ?str, ?cit)$
$U\ myaddress(?num, ?str, ?cit)$
$if \quad ?pr < 400 \quad \wedge \quad war(?s1, france) \not\ll ?P$
}

7 Benchmarking

In this section we benchmark several languages for production rule systems. For this, we use the fibonacci numbers sequence, encoded as following (independent of the resolution strategy):

```
P := { false/0, true/0, not/1, >/2, remove/1, add/1 }
F := { fib/2 }
W0 := { fib(0, 1),  fib(1, 1),  fib(200, -1) }

R := {
  if
    fib(?n, -1),
    not fib(?n-1, ?v)
  then
    add(fib(?n-1, -1))
  end if

  if
    fib(?n, -1),
    fib(?n-1, ?v1>0),
    fib(?n-2, ?v2>0)
  then
    remove(fib(?n, -1)),
    add(fib(?n, ?v1+?v2)),
    remove(fib(?n-2, ?v2))
  end if

  if
    fib(200, ?v>0),
    fib(199, ?v1)
  then
    remove(fib(199, ?v1))
  end if
}
```

The benchmark has been executed on a Pentium IV 1.70Ghz 256Kb L2-cache and 384 MB RAM running Mandrake 9.2, see tables 1 to 4:

Language	Time [ms]	Rules fired	Rules/sec
Clips 6.1	40 ms	398 pr	9925 pr/sec
JeOPS 2.1	215 ms	398 pr	1847 pr/sec
JRules 6.0	686 ms	398[6] pr	580 pr/sec
ELAN 3.6g	810 ms	598 rwr	738 rwr/sec
JTom 2.0rc2	311 ms	398 pr	1276 pr/sec

Table 1: Fibonacci(200)

Language	Time [ms]	Rules fired	Rules/sec
Clips 6.1	160 ms	798 pr	4988 pr/sec
JeOPS 2.1	482 ms	798 pr	1654 pr/sec
JRules 6.0	1224 ms	798[7] pr	651 pr/sec
ELAN 3.6g	6590 ms	1197 rwr	181 rwr/sec
JTom 2.0rc2	1457 ms	798 pr	547 pr/sec

Table 2: Fibonacci(400)

7.1 A more demanding benchmark

The same fibonacci sequence, but without *garbage collection*.

Language	Time [ms]	Rules fired	Rules/sec
Clips 6.1	1010 ms	1997 pr	1977 pr/sec
JeOPS 2.1	903 ms	1997 pr	2211 pr/sec
JRules 6.0	4332 ms	1997[8] pr	460 pr/sec
JTom 2.0rc2	18557 ms	1997 pr	107 pr/sec

Table 3: Fibonacci(1000)

Language	Time [ms]	Rules fired	Rules/sec
Clips 6.1	163070 ms	19997 pr	123 pr/sec
JeOPS 2.1	59938 ms	19997 pr	334 pr/sec
JRules 6.0	259022 ms	19997[9] pr	77 pr/sec

Table 4: Fibonacci(10000)

```
P := { false/0, true/0, not/1, >/2, remove/1, add/1 }
F := { fib/2 }
W0 := { fib(0, 1),  fib(1, 1),  fib(200, -1) }

R := {
  if
    fib(?n, -1),
    not fib(?n-1, ?v)
  then
    add(fib(?n-1, -1))
  end if

  if
    fib(?n, -1),
    fib(?n-1, ?v1>0),
    fib(?n-2, ?v2>0)
  then
    remove(fib(?n, -1)),
    add(fib(?n, ?v1+?v2)),
  end if
}
```

The results are:

Language	Time [ms]	Rules fired	Rules/sec
Clips 6.1	10 ms	197 pr	19700 pr/sec
JeOPS 2.1	244 ms	197 pr	1428 pr/sec
JRules 6.0	966 ms	197[10] pr	204 pr/sec
ELAN 3.6g	83780 ms	287 rwr	3 rwr/sec

Table 5: Fibonacci(100)

Language	Time [ms]	Rules fired	Rules/sec
Clips 6.1	60 ms	397 pr	6617 pr/sec
JeOPS 2.1	370 ms	397 pr	1588 pr/sec
JRules 6.0	1171 ms	397 [11] pr	339 pr/sec
ELAN 3.6g	2454680 ms	597 rwr	0.24 rwr/sec

Table 6: Fibonacci(200)

References

[BBCK04] Clara Bertolissi, Paolo Baldan, Horatiu Cirstea, and Claude Kirchner. A rewriting calculus for cyclic higher-order term graphs. In Maribel Fernandez, editor, *Proceedings of the 2nd International Workshop on Term Graph Rewriting*, Roma (Italy), September 2004. Electronic Notes in Theoretical Computer Science. to appear.

[BCD+04] Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Quang-Huy Nguyen, Christophe Ringeissen, and Marian Vittek. ELAN *V 3.6 User Manual*. LORIA, Nancy (France), fifth edition, February 2004.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[CFK04] Horatiu Cirstea, Germain Faure, and Claude Kirchner. A ρ-calculus of explicit constraint applications. In Narciso Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, Electronic Notes in Theoretical Computer Science, pages 47–63. Elsevier, 2004.

[CK01] Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[CKLW03] Horatiu Cirstea, Claude Kirchner, Luigi Liquori, and Benjamin Wack. Rewrite strategies in the rewriting calculus. In Bernhard Gramlich and Salvador Lucas, editors, *Third International Workshop on Reduction Strategies in Rewriting and Programming* , Valencia, Spain, June 2003. Electronic Notes in Theoretical Computer Science.

[CKMM04] Horatiu Cirstea, Claude Kirchner, Michael Moossen, and Pierre-Etienne Moreau. Production systems and rete algorithm formalisation. Manifico deliverable, LORIA, Nancy, September 2004.

[Com91] H. Comon. Disunification: a survey. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 9, pages 322–359. The MIT press, Cambridge (MA, USA), 1991.

[DK99a] Hubert Dubois and Hélène Kirchner. Modelling planning problems with rules and strategies. Technical Report 99-R-029, LORIA, Nancy, France, March 1999.

[DK99b] Hubert Dubois and Hélène Kirchner. Rule based programming with constraints and strategies. Technical Report 99-R-084, LORIA, Nancy, France, November 1999. ERCIM workshop on Constraints, Paphos (Cyprus).

[DK00a] Hubert Dubois and Hélène Kirchner. Objects, rules and strategies in ELAN. In *Proceedings of the second AMAST workshop on Algebraic Methods in Language Processing, Iowa City, Iowa, USA*, May 2000.

[DK00b] Hubert Dubois and Hélène Kirchner. Rule Based Programming with Constraints and Strategies. In K.R. Apt, A.C. A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus, October 25-27, 1999*, volume 1865 of *Lecture Notes in Artificial Intelligence*, pages 274–297. Springer-Verlag, 2000.

[Dub01] Hubert Dubois. *Systèmes de règles de production et calcul de réécriture*. PhD thesis, Université Henri Poincaré - Nancy 1, September 2001.

[Eke95] Steven Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

[FH86] F. Fages and G. Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43(1):189–200, 1986.

[FK02] Germain Faure and Claude Kirchner. Exceptions in the rewriting calculus. In Sophie Tison, editor, *Proceedings of the RTA conference*, Lecture Notes in Computer Science, page pp, Copenhagen, July 2002. Springer-Verlag.

[FL91] Francois Fages and Rémi Lissajoux. Systèmes experts temps-réel: une introduction au langage xrete. *Revue Technique Thomson-CSF*, 23 - 3:633–699, 1991.

[FL92] Francois Fages and Rémi Lissajoux. Sémantique opérationnelle et compilation des systèmes de production. *Revue d'intelligence artificielle*, 6 - 4:431–456, 1992.

[For81] Charles Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, USA, July 1981. 62 pages.

[For82] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Programming*, 37(1-3):98–135, October 1998.

[Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*, volume 5 of *Computer Science and Technology Series*. Harper & Row, New York, 1986.

[Hul79] J.-M. Hullot. Associative-commutative pattern matching. In *Proceedings 9th International Joint Conference on Artificial Intelligence*, 1979.

[Hul80] J.-M. Hullot. *Compilation de Formes Canoniques dans les Théories équationelles*. Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France), 1980.

[KB70] Donald E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[KDK93] Francis Klay, Eric Domenjoud, and Claude Kirchner. Vérification sémantique de spécifications métallurgiques. Rapport de fin de contrat, Inria Lorraine & Crin, 1993.

[KK99] Claude Kirchner and Hélène Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KKV94] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Implementing computational systems with constraints. In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *Proceedings of the first PPCP workshop*. The MIT press, 1994. To appear.

[KM01] Hélène Kirchner and Pierre-Etienne Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001. Report LORIA A01-R-007.

[LG89] Thomas Laffey and Anoop Gupta. Real-time knownledge-based systems, 1989.

[Lop87] Frank Lopez. *The Parallel Production System*. PhD thesis, University of Illinois, Urbana-Champaign, Illinois, USA, 1987.

[MBLG90] Daniel Miranker, David Brant, Bernie Lofaso, and David Gadbois. On the performance of lazy matching in production systems. *Knowledge Presentation*, pages 685–692, 1990.

[Mir90] Daniel Miranker. *Treat: a new and efficient match algorithm for AI production systems*. Morgan Kaufmann Publishers, 1990.

[PS81] G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.

[RN93] A. Rubio and R. Nieuwenhuis. A precedence-based total ac-compatible ordering. In C. Kirchner, editor, *Proceedings 5th Conference on Rewriting Techniques and Applications, Montreal (Canada)*, volume 690 of *Lecture Notes in Computer Science*, pages 374–388. Springer-Verlag, 1993.

[Rub02] Alberto Rubio. A fully syntactic ac-rpo. 178(2):515–533, November 2002.

[SS96] Wayne Snyder and James Schmolze. Rewrite semantics for production rule systems: Theory and applications. In Michael McRobbie and John Slaney, editors, *Proceedings 13th International Conference on Automated Deduction, New Brunswick NY (USA)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 508–522. Springer-Verlag, July 1996.

[Sti75] M. E. Stickel. A complete unification algorithm for associative-commutative functions. In *Proceedings 4th International Joint Conference on Artificial Intelligence, Tbilissi (USSR)*, pages 71–76, 1975.

[“T02] “Terese” (M. Bezem, J. W. Klop and R. de Vrijer, eds.). *Term Rewriting Systems*. Cambridge University Press, 2002.

[X-T88] X-tra 1.0 - manuel de reference, 1988.